BRANCH-AND-CUT SOLUTION OF INFERENCE PROBLEMS IN PROPOSITIONAL LOGIC

J.N. HOOKER *

Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA 15213, USA

C. FEDJKI *

Ecole El Ghazali, Rue Suidani Boudjemaa, Constatine 25000, Algeria

Abstract

We describe and test computationally a branch-and-cut algorithm for solving inference problems in propositional logic. The problem is written as an integer program whose variables correspond to atomic propositions. We generate cuts for the integer program using a separation algorithm based on the resolution method for theorem proving. We find that the algorithm substantially reduces the size of the search tree when it is large. It is faster than Jeroslow and Wang's method on hard problems and slower on easy problems.

Keywords: Propositional logic, cutting planes, integer programming.

1. Introduction

The inference problem in propositional logic is notoriously hard. It is equivalent to the satisfiability problem, which was the original NP-complete problem [2]. It has traditionally been solved by nonnumeric methods, such as the various resolution procedures [7,13]. But recent evidence indicates that mathematical programming methods provide one of the more promising approaches for solving hard inference problems. These methods are based on the fact that an inference or satisfiability problem can be written as integer program that can in turn be solved with methods that exploit its structure. Our purpose here is to test the effectiveness of a particular method for solving the integer program, namely a branch-and-cut method that uses a new separation algorithm based on a connection between resolution and cutting plane theory.

The two best-known methods for solving integer programs, branch-and-bound and cutting plane methods, have been applied to the integer program underlying an inference problem. The branch-and-bound technique was used by Blair, Jeroslow and Lowe [1], who pioneered the mathematical programming approach

* Supported in part by the US Air Force Office of Scientific Research, grant AFOSR-87-0292.

to inference in propositional logic. They showed that a straightforward branch-and-bound algorithm can solve a large class of random problems by enumerating only a few nodes in the search tree. Their motivating idea is that by solving the linear programming (LP) relaxation of the problem at each node of the branch-and-bound tree, they may happen upon an integer solution early in the process and thereby solve the problem quickly. Moreover, the objective function in the integer program directs the search for a solution in a way that is not available in such symbolic methods as the Davis–Putnam procedure with "splitting" [3,13], which is the nonnumeric analogue of branch-and-bound for satisfiability problems.

The cutting plane approach has been applied by one of us (Hooker), who exploited the fact that a resolvent is a particular type of "rank 1" cutting plane [8,10]. This means that resolution can be viewed as a technique that generates cutting planes for solving the underlying integer program. Since resolution tends to generate a very large number of formulas, nonnumeric resolution methods must often use heuristics to select which resolvents to generate. The integer programming context, however, provides a natural guide for generating only a few useful resolvents (namely, those that are "separating" cuts, which we define below). Computational tests of a method based on this principle showed that resolution cuts, with occasional recourse to branch-and-bound, can check whether a conclusion follows from a set of 40–100 randomly generated premises more than 1000 times more rapidly than resolution when the conclusion does not follow, and about twice as rapidly otherwise. (In the latter case only two or three premises are generally needed to obtain the conclusion, so that resolution does not lead to an exponential explosion.) In some instances, however, no resolution cut was separating, and this happens quite often in the more difficult problems we generated for the present study. It is therefore useful to have stronger cuts.

Both branch-and-bound and cutting plane approaches require the costly solution of highly degenerate LP relaxations. Jeroslow and Wang addressed this problem by replacing a relatively sluggish LP routine with a heuristic for generating integer solutions ([12], also presented in [9]). The resulting algorithm was an order of magnitude faster than LP-based branch-and-bound. This raises doubt as to whether an LP-based algorithm can ever compete effectively with techniques in which a nonnumeric algorithm replaces the LP. Our purpose is to try to determine whether this doubt is warranted.

In this paper we do two things: (1) we combine the branch-and-bound and cutting plane approaches to obtain a branch-and-cut algorithm, and (2) we enlarge the class of cutting planes so as to increase the probability that a separating cut can be found with a reasonable amount of computation. We test the resulting algorithm computationally and compare it with a pure branch-and-bound approach as well as the Jeroslow–Wang method. The problems are randomly generated in such a way that many are quite hard, much harder than those studied in [10].

We find that the branch-and-cut method requires substantially less time on random problems than branch-and-bound when the latter must enumerate a relatively large number of nodes, which indicates that our cutting planes are useful in the harder problems where they are needed most. We also find that the branch-and-cut method is superior to the Jeroslow–Wang method on hard problems (those requiring a large number of branch-and-bound nodes), whereas the Jeroslow–Wang method is better on relatively easy problems. This suggests that when the problem is hard, the substantial overhead of solving an LP relaxation is normally more than offset by the direction-finding capability provided by an objective function and separating cuts. We do not compare branch-and-cut with resolution-based methods, since our experience in [10] indicates that the latter would run far too slowly to solve problems of the size solved here.

Our method for generating cuts is suggested by a resolution-based separation algorithm developed by one of us (Hooker, [11]). This earlier algorithm generates a clause that is a rank one separating cut if such a cut exists, but it can be impractical to use. Here we modify the algorithm to make it more practical and prove that it nonetheless generates any separating cut that the original algorithm does, plus perhaps some additional separating cuts.

2. Logical fundamentals

The formulas of *propositional logic* consist of *atomic propositions* or *variables* x_j joined by such connectives as *and, or, not*, and *implies*. A *literal* is an atomic proposition x_j or its *negation* $\neg x_j$. A *clause* is a disjunction of literals, such as

$$x_1 \vee \neg x_2 \vee x_3. \tag{1}$$

One loses no generality by dealing only with clauses, since any formula is equivalent to a conjunction of clauses [7]. A *unit clause* contains exactly one literal, and the *empty clause* contains no literals and is necessarily false.

A *truth assignment* is an assignment of *true* or *false* to every variable. A set S of clauses is *satisfiable* if some truth assignment makes every clause in S true. S *implies* a clause C if every truth assignment that makes all the clauses in S true makes C true. It is not hard to see that clause C implies clause D if and only if C *absorbs* D; that is, every literal of C occurs in D.

If exactly one variable x_j occurs negated in one clause and unnegated in another, the two clauses have as their *resolvent on* x_j the clause containing all literals occurring in either parent, except x_j and $\neg x_j$. For instance, (3) below is the resolvent of (1) and (2).

$$\neg x_1 \vee \neg x_2 \vee \neg x_4, \tag{2}$$

$$\neg x_2 \vee x_3 \vee \neg x_4. \tag{3}$$

The resolvent is implied by its parents jointly but by neither individually.

A *resolution proof* of a clause C from a set of premises is a finite sequence of clauses beginning with the premises and ending with C, such that every clause is the resolvent of two earlier clauses in the sequence. Let us say that a clause in a resolution proof is a *unit support clause* if it contains exactly one variable that does not appear in the conclusion C. A *unit support proof* of C, as defined in [11], is a resolution proof in which at least one parent of every resolvent is a unit support clause.

3. Integer programming fundamentals

Any clause can be written as a linear inequality in binary variables. For instance, (1) can be written,

$$x_1 + (1 - x_2) + x_3 \geqslant 1 \quad \text{or } x_1 - x_2 + x_3 \geqslant 0,$$

where each $x_j \in \{0, 1\}$. We interpret $x_j = 1$ to mean x_j is true, and $x_j = 0$ to mean x_j is false. Let a set S of m clauses in n variables be written as an $m \times n$ system $Ax \geqslant a$ of linear inequalities, where each a_i is equal to 1 minus the number of -1's in row i of A. Clearly, S is satisfiable if and only if the minimum value of the artificial variable x_0 is one in the following integer program,

$$\begin{array}{ll} \text{minimize} & x_0 \\ \text{subject to} & x_0 e + Ax \geqslant a, \quad \text{all } x_j \in \{0, 1\}, \end{array} \tag{4}$$

where e is a column of m ones. We can also determine whether a set S' of clauses implies a clause C by solving a problem having the form of (4). For each literal x_j (or $\neg x_j$) of C, augment the linear system representing S' with that literal's denial, namely the inequality $-x_j \geqslant 0$ (or $x_j \geqslant 1$). If $Ax \geqslant a$ is the augmented system, then S' implies C if and only if the minimum value of x_0 in (4) is one. Since we can thus solve inference problems by solving satisfiability problems, we will concern ourselves only with the latter.

Consider the following *linear programming relaxation* of (4):

$$\begin{array}{ll} \text{minimize} & x_0 \\ \text{subject to} & x_0 e + Az \geqslant 1, \quad 0 \leqslant x_j \leqslant 1 \quad \text{for all } j. \end{array} \tag{5}$$

Let (x_0^*, x^*) be a (possibly nonintegral) optimal solution of (5), if one exists. Three cases are relevant to solving the satisfiability problem that (4) represents: (a) $x_0^* > 0$, which means already that S is unsatisfiable, since any solution (x_0, x) of (4) must have $x_0 \geqslant x_0^*$ and therefore $x_0 = 1$ (since x_0 is integral); (b) $x_0^* = 0$ and x^* is integral, in which case S is satisfiable; (c) $x_0^* = 0$ and x is not integral, in which case S may or may not be satisfiable.

A *cut* for (5) is an inequality satisfied by all feasible solutions of (4). A *separating cut* for a solution (x_0^*, x^*) of (5) is a cut that (x_0^*, x^*) violates. A *cutting plane* approach to solving the satisfiability problem would be to add one or more separating cuts to (5), solve the resulting problem, add more cuts if necessary, and so on until case (a) or (b) is obtained.

In a *branch-and-bound* approach to solving (4), the optimal solution x^* of the LP relaxation (5) is regarded as the root node of a search tree. One then *branches* on a variable x_j with a fractional value x_j^* by first re-solving (5) with the additional constraint $x_j = 0$ and then re-solving (5) with the additional constraint $x_j = 1$. This creates two successor nodes of the root node, at each of which the procedure is repeated in a recursive fashion. A node is *fathomed* (i.e., all successor nodes pruned) when case (b) occurs, at which point the search backtracks. The algorithm terminates with satisfiability when case (c) is obtained at a node, or with unsatisfiability when the search is completed without obtaining case (b).

A *branch-and-cut* algorithm simply generates cuts at every node of the search tree.

A cut belongs to the *elementary closure* of a linear system $Ax \geqslant a$ if it is the result of taking a positive linear combination of inequalities in the system and rounding up any nonintegers that result. A cut is a *rank 1* cut if it is a positive linear combination of inequalities in the elementary closure.

The resolvent of two clauses is easily seen to be a cut, which we call a *resolution cut*. In fact, it is a rank 1 cut with respect to the inequalities that the clauses represent (and bounds of the form $0 \leqslant x_j \leqslant 1$). For instance, the resolvent (3) of (1) and (2) can be obtained by taking a weighted sum in which inequalities representing (1) and (2) and the bounds $x_3 \geqslant 0$ and $-x_4 \geqslant -1$ all get weight 1/2. The result is $-x_2 + x_3 - x_4 \geqslant -3/2$, in which the right hand side can be rounded up to obtain an inequality representing the resolvent (3). Not all clauses that are rank 1 cuts, however, need be resolvents of the original premises.

4. The problem of finding separating cuts

Since resolvents are cuts, one way to obtain separating cuts is to generate resolvents of the clauses in (4) and retain those that are separating cuts. But it was found in [10] that in many cases, none of these resolvents are separating. In this event one might use resolvents as parents to generate still more resolvents, and so on repeatedly, until finding a separating cut. But since the number of resolvents tends to grow explosively, this is not a practical approach.

In [11] one of us (Hooker) proposed reducing the number of resolvents by generating only clauses that are *rank 1* cuts. It was proved in [11] that we can obtain all such cuts by using unit support proofs alone, which do not produce nearly so many resolvents as general resolution proofs. In fact, [11] shows that the following algorithm generates a clause that is a rank 1 separating cut if one exists. Given a solution (x_0^*, x^*) of (5), let the *truth value* of literal x_j be x_j^* and of $\neg x_j$ be $1 - x_j^*$.

4.1. SEPARATION ALGORITHM 1

Step 0. Let S be the set of clauses represented by $Ax \geqslant a$ in (4), and solve the LP relaxation (5). Set $k = 0$.
Step 1. Set $k = k + 1$. If $k > n$, stop.
Step 2. Set $T = S$, and pick a set K of k variables (a set not already picked) with fractional values in the solution of (5). If no such sets remain, go to Step 1.
Step 3 (*truth value condition*). Pick a clause D in T with exactly one variable x_j in K, such that the sum of the truth values of the other literals in D is strictly less than 1. If there is no such D, go to Step 2.
Step 4 (*unit support resolution*). Derive all possible resolvents on x_j of D with other clauses in T, and add the resolvents to T. Go to Step 3, unless one of these resolvents is separating (and no more separating cuts are desired), in which case the algorithm stops.

Yet even this algorithm can be impractical, since a prohibitively large number of sets K may need to be enumerated (even if we terminate the algorithm before k becomes large). We can obtain a more practical algorithm by dropping the set K and accumulating resolvents rather than resetting T to S in Step 2. We retain the truth value conditions in Step 3, however, and they continue to limit the production of resolvents. We will show that the resulting algorithm, below, generates any separating cut that Algorithm 1 does and thus in particular generates a rank 1 separating cut if such a clause exists.

4.2. SEPARATION ALGORITHM 2

Step 0. Let S be the set of clauses $Ax \geqslant a$ in (4), and solve the LP relaxation (5). Set $k = 0$.
Step 1. Set $k = k + 1$. If $k > n$, stop.
Step 2 (*truth value condition*). Pick from S a clause D containing a variable with a fractional value in the solution of (5), such that the sum of the truth variables of the other variables in D is strictly less than 1. If there is no such D, go to Step 1.
Step 3 (*resolution*). Derive all possible resolvents of D and other clauses in S that take place on a variable x_j with a fractional value in the solution of (5). Add the resolvents to S. Go to Step 2, unless one of these resolvents is separating (and no more separating cuts are desired), in which case the algorithm stops.

To illustrate the algorithm, suppose that (6)–(10) below are among the constraints in (4). The solution x^* of (5) is shown in the first row.

$$\begin{array}{lllllll} x^* = 0 & 1/5 & 2/5 & 1/5 & 4/5 & 2/5 & 2/5 \end{array}$$

$$x_0 + x_1 + x_3 + x_4 + x_5 \geqslant 1, \tag{6}$$

$$x_0 + x_1 - x_3 + x_5 \geqslant 0, \tag{7}$$

$$x_0 - x_1 + x_2 + x_5 + x_6 \geqslant 0, \tag{8}$$

$$x_0 - x_2 + x_4 + x_6 \geqslant 0, \tag{9}$$

$$x_0 - x_4 + x_6 \geqslant 0. \tag{10}$$

Since the truth values of literals other than $-x_3$ in (7) sum to 3/5, which is less than 1, we can resolve (7) and (6) to obtain (11) below. Similarly, we resolved (6) and (10) to obtain (12), and (9) and (10) to obtain (13).

$$x_0 + x_1 + x_4 + x_5 \geqslant 1, \tag{11}$$

$$x_0 + x_1 + x_3 + x_5 + x_6 \geqslant 1, \tag{12}$$

$$x_0 - x_2 + x_6 \geqslant 0. \tag{13}$$

We can now resolve (13) and (8) to obtain (14) below, and (11) and (10) to obtain (15).

$$x_0 - x_1 + x_5 + x_6 \geqslant 0, \tag{14}$$

$$x_0 + x_1 + x_5 + x_6 \geqslant 1. \tag{15}$$

These two resolve to produce,

$$x_0 + x_5 + x_6 \geqslant 1, \tag{16}$$

which is a separating cut. It is a rank 1 cut, as can be verified by taking a linear combination in which (6)–(10) and the bound $x_5 \geqslant 0$ respectively have weights 1/7, 1/7, 2/7, 2/7, 3/7, 3/7. By comparison, Algorithm 1 must enumerate 15 test K and generate 33 resolvents before discovering a separating cut, namely (16), when $K = \{x_1, x_2, x_3, x_4\}$.

We now show that Algorithm 2 is at least as powerful as Algorithm 1.

THEOREM 1

Algorithm 2 finds all separating cuts found by Algorithm 1 and therefore generates a clause that is a rank 1 cut if one exists.

Proof

Let Algorithm 1 or 2 *generate* a clause C if $C \in S$ or C is a resolvent obtained in the algorithm. It suffices to show that Algorithm 2 generates all the clauses that Algorithm 1 does. Let a resolvent obtained in Algorithm 1 have *depth* $d+1$ if the maximum depth of its parents is d, where all clauses in S are assigned depth zero. The proof is by induction on depth.

Obviously, Algorithm 1 generates all clauses of depth zero that Algorithm 2 does. Now, supposing that Algorithm 2 generates all clauses of depth d that Algorithm 1 does, we will prove the same for depth $d+1$.

Let R be any resolvent of depth $d+1$ that Algorithm 1 generates. Let D^* be the clause D that Algorithm 1 uses in Step 4 to obtain R, and E^* a clause with which the algorithm resolves D^* to obtain R. Then D^* and E^* have depth d or less. Also D^* contains exactly one variable x_j in K, and the sum of the truth values of the variables in D^* other than x_j is strictly less than 1. By the induction hypothesis Algorithm 2 generates D^* and E^*, which at some point have been added to S in Step 3. Thus in some execution of Step 2, Algorithm 2 picks D^* as the clause D and resolves it on x_j with E^*, thus obtaining R. □

Algorithm 2 can also generate cuts that have rank greater than 1. Given the clauses,

$$x_0 + x_1 + x_2 + x_3 \geqslant 1,$$

$$x_0 + x_1 - x_2 + x_3 \geqslant 0,$$

$$x_0 - x_1 + x_2 + x_3 \geqslant 0,$$

$$x_0 - x_1 - x_2 + x_3 \geqslant -1,$$

and supposing $(x_0^*, \ldots, x_3^*) = (0, 1/2, 1/2, 0)$, Algorithm 2 yields the separating cut $x_0 + x_3 \geqslant 1$, which is not a rank 1 cut and is not produced by Algorithm 1.

5. A branch-and-cut algorithm

We now present a branch-and-cut algorithm that exploits the special properties of the satisfiability problem. The algorithm makes essential use of *unit resolution*, which repeatedly looks for a unit clause, and removes all clauses that contain the literal in the unit clause, as well as all occurrences of the literal's negation in the remaining clauses. The procedure terminates when no unit clauses remain or the empty clause is generated. In the latter case, we know the original set of clauses was unsatisfiable.

The branch-and-cut algorithm is as follows. At any node of the branch-and-bound tree, let (4) be the original satisfiability problem, plus all cuts generated at predecessor nodes. We first apply unit resolution to (4). If the empty clause results, we know (4) is inconsistent, and we fathom the current node and backtrack. Otherwise we compute the solution (x_0^*, x^*) of the LP relaxation (5) of (4). If $x_0^* > 0$ then (4) is inconsistent, and we fathom the node and backtrack. If $x_0^* = 0$ and x^* is integral, the problem is satisfiable, and we can stop.

If we can neither stop nor fathom the current node, we use our separation algorithm to generate some cuts, add the cuts to (4), and re-solve the relaxation (5) for (x_0^*, x^*). Again we backtrack if $x_0^* > 0$ and stop if we find a satisfying solution. Otherwise we use a heuristic (described below) to pick a fractional variable x_j^* on which to branch, and to pick which branch to explore first. If we have already branched at the current node, however, we simply take the unexplored branch, unless both branches have been explored, in which case we fathom the node and backtrack. If we branch we create a new node, to which we apply the same procedure. If we fathom every node without finding a satisfying solution, the problem is unsatisfiable, and otherwise it is satisfiable.

Note that the problem (4) tends to shrink as we move deeper into the tree, since more variables are fixed at the deeper nodes, which allows the unit resolution step to eliminate more clauses and literals. This contrasts favorably with most branch-and-cut procedures, in which the problem grows with depth. But we must either store the problem associated with each node at which we branch, or regenerate it when we backtrack to the node. We choose the latter option, since it requires only that we add to the original problem the fixed variables and the cuts generated at the node's predecessors, and apply unit resolution.

We use the heuristic of Jeroslow and Wang to choose the variable on which to branch. If S is the set of clauses represented by the constraints of (4), define a weight function $w(S) = \Sigma_k N(k) 2^{-k}$, where k ranges over positive integers and $N(k)$ is the number of clauses in S containing k literals. If $S(j, v)$ is the set of clauses of S in which x_j occurs (for $v = 1$) or in which $\neg x_j$ occurs (for $v = 0$), let j^* and v^* be arguments that maximize $w(S(j, v))$ over all j such that x_j^* is fractional. We then branch on x_{j^*} by first assigning it the value v^*.

The LP relaxation tends to be highly degenerate and therefore difficult to solve. We found that we can accelerate the solution substantially with a simple stepwise process. We first solve (5) using only the constraints with no negative coefficients. We add the violated constraints and re-solve, and continue in this fashion until all constraints are satisfied. Each time we add violated constraints or new cuts we start with the previous optimal basis, augmented with surplus variables as necessary.

Finally, our separation algorithm (Algorithm 2) allows several practical improvements. Let us refer to the value of k as the number of *passes* that have been executed. To keep execution time reasonable we use at most $k_{\max}$ $(< n)$ passes. We also observe that prior to the last pass there is no point in saving nonseparating resolvents in Step 3 that contain no variables with fractional values, since they cannot be used as parents of further resolvents. Furthermore, if the truth value of a clause is the sum of the truth values of its literals, then the truth value of a parent cannot exceed that of its resolvent by more than 1. Since we want the resolvents obtained in the last pass to be separating (i.e., to have truth value less than 1), the resolution parents we consider in pass k should have truth value less than $k_{\max} - k + 2$.

The more effective cuts are generally those that contain fewer literals, particularly those that are unit clauses. We found empirically that these tend to be generated in the later passes. But the algorithm often generates an unmanageable number of cuts before the later passes are reached, so that it must be prematurely terminated before obtaining the best cuts. To solve this problem we permitted only unit resolvents to be generated in the last pass. Since the length of a parent cannot exceed that of its resolvent by more than 1, this permits us to consider in pass k only parents of length at most $k_{\max} - k + 2$. We also place an upper bound N on the number of cuts to be generated on each pass.

The improved separation algorithm follows.

5.1. SEPARATION ALGORITHM 2′

Step 0. Let $T = S'$, $k = 0$, $C = \varnothing$.

Step 1. Set $k = k + 1$. If $k > k_{\max}$, stop.

Step 2 (*truth value condition*). Pick from T a clause D of length at most $k_{\max} - k + 2$ that contains a variable with a fractional variable x_j^*, such that the sum of the truth values of the other variables in D is strictly less than 1. If there is no such D, go to Step 1.

Step 3 (*resolution*). Derive all possible resolvents of D with other clauses E in T that take place on a variable with a fractional value x_j^*, where E has length at most $k_{\max} - k + 2$ and truth value strictly less than $k_{\max} - k + 2$. Add to T the resolvents containing at least one variable with a fractional value, until the cardinality of T reaches N. Add to (4) the resolvents that are separating. If more separating resolvents are desired, go to Step 2; otherwise stop.

6. Implementation

The branch-and-cut algorithm was written in Microsoft FORTRAN 4.1. The LP relaxations were solved by calls to Marsten's XMP [14]. The data structure for the set S of clauses consisted of two linked lists for each clause, one containing the indices of the positive literals and one those of the negative literals. To perform the resolutions in Step 3 of the separation algorithm, two additional linked lists were used for each clause: one of positive literals with fractional truth values, and one of negative literals with fractional truth values. This accelerates checking whether two clauses can be resolved on a variable with a fractional value.

7. Problem generation

Two probability models are generally used for generating random satisfiability problems in n variables [5,6,16].

Fixed density model. Build each clause by randomly choosing k distinct variables from $\{x_1, \ldots, x_n\}$ and negating each with probability $1/2$.

Fixed probability model. Build each clause by letting each variable x_j appear in the clause with probability p, and negate each variable that appears with probability 1/2. If the resulting clause is empty, generate another.

The fixed density model is highly unrealistic. It is hard to imagine an application, in either artificial intelligence or computer engineering, in which the number of literals per clause would be constant. It is reasonable, however, that the *expected* number of literals per clause would be more or less independent of the number of clauses or variables. The length of propositions in large knowledge bases, for instance, would probably be about the same as in small ones. This suggests that we should use the fixed probability model and set the expected number np of literals per clause to a constant, perhaps between 3 and 10. We choose $np = 5$.

One difficulty with the fixed probability model, however, is that a set of m clauses will generally contain several unit clauses. Let $b_n(k)$ be the binomial probability $b_n(k) = \binom{n}{k} p^k (1-p)^{n-k}$. The expected number of unit clauses is $mb_n(1)/(1 - b_n(0))$, or 3% of the clause for a problem typical of our test problems ($n = 100$, $np = 5$). The presence of unit clauses permits one to simplify the problem, often substantially, by preprocessing it with unit resolution, so that the residual problems actually solved with a branch-and-cut method have widely varying sizes. To remove this source of variability, we exclude unit clauses and keep generating random clauses until m nonunit clauses are obtained.

Another difficulty with the fixed probability model is that satisfiable problems tend to have a very large number of solutions and therefore to be easy to solve, since a solution can often be found with relatively little backtracking. The probability that a given binary vector x satisfies a random clause containing k literals is $1 - (1/2)^k$. Since the probability that a nonunit clause C contains k literals ($k \geqslant 2$) is the truncated binomial probability $\bar{b}_n(k) = b_n(k)/(1 - b_n(0) - b_n(1))$, the probability that x satisfies C is $P_n = \sum_{k=2}^{n} (1 - (1/2)^k) \bar{b}_n(k)$. So, the probability that x satisfies m clauses is P_n^m, and the expected number of solutions is $2^n P_n^m$. For $n = 100$ and $np = 5$, a problem with 740 clauses has an expected number of more than 10^{10} solutions, and the number is much larger for smaller problems. Conversely, a problem with 1200 clauses has an expected number of 0.05 solutions and is therefore almost certainly unsatisfiable, and the number drops precipitously for larger problems. We will find in fact that nearly all problems with 900 clauses are unsatisfiable. This suggests that satisfiable problems are likely to be easy except when the number of clauses begins to approach 900, at which point satisfiable problems become scarce.

Unsatisfiable problems also tend to be easy on the fixed probability model, because when there are enough clauses to make unsatisfiability probable, assigning values to only a few variables is likely to falsify all the literals in some clause. When this happens it may be unnecessary to probe deeply into the search tree to verify unsatisfiability. The probability $\bar{P}_n(i)$ that a partial vector $(x_1, \ldots, x_i)$ falsifies all the literals in a nonunit clause C is the probability $(1-p)^{n-i}$ that all of C's literals have received a truth value times the probability they are all false. Thus $\bar{P}_n(i) = (1-p)^{n-i}\Sigma_{k=2}^{i}(1-(1/2)^k)\bar{b}_i(k)$. When $n = 100$ and $np = 5$ the probability that fixing, say, five variables will falsify one of 700 clauses is about 0.73, but the probability it will falsify one of 2000 clauses is 0.98. We should therefore expect unsatisfiable problems to become steadily easier as the number of clauses increases. These considerations suggest that the hardest problems should occur as m increases to the point where the preponderance of problems shifts from satisfiable to unsatisfiable. The shift occurs fairly rapidly due to the precipitous drop in the expected number of solutions. We therefore fix n at 100 and let m range from somewhat below 900 to somewhat above 900. We also do a similar experiment for $n = 50$.

8. Computational results

The FORTRAN routines were run on an IBM PS/2 Model 80 personal computer using the OS/2 Version 1.1 operating system. We permitted a maximum of three passes ($k_{\max} = 3$) in the separation algorithm and a maximum of 200 cuts to be generated at each pass ($N = 200$).

Table 1 summarizes the results for random problems having 50 and 100 variables. The branch-and-cut method is compared with a pure branch-and-bound method (the branch-and-cut algorithm with Step 3 omitted). It is also compared

Table 1
Computational results. Computer times are minutes on an IBM PS-2 Model 80 personal computer

No. rows	No. problems	No. satisfiable	Branch & bound		Jeroslow & Wang		Branch & cut	
			Avg. no. nodes	Avg. time	Avg. no. nodes	Avg. time	Avg. no. nodes	Avg. time
100 variables								
700	10	10	107	91	61	13	29	43
800	20	11	138	169	219	138	34	77
850 [a)]	20	4	144	206	160	117	28	69
900	10	1	136	213	116	93	34	93
1000	10	0	30	64	33	32	7.0	28
1500	10	0	9.4	39	9.4	16	3.8	22
50 variables								
300	10	10	12	1.6	25	0.3	11	2.3
400	10	8	14	3.4	22	1.6	10	4.5
450	10	4	22	8.4	25	4.4	10	7.1
500	10	0	13	5.9	12	3.1	4.6	4.5
750	10	0	3.9	3.8	4.1	2.0	1.0	1.9

[a)] To avoid obscuring the typical pattern, these averages (and the two plots) omit one unusually hard problem for which the three methods respectively searched 1823, 1807 and 471 nodes and required 1007, 1422 and 1078 minutes of computation time.

Table 2
Cuts generated in the 80 problems having 100 variables

Number of rows:	700	800	850	900	1000	1500	Total
% of nodes requiring cuts	84	72	61	60	56	55	67
Of those nodes requiring cuts:							
% for which cuts were found	70	90	91	95	97	95	92
% for which pass 3 cuts were found [a]	30	50	53	54	64	62	55
% for which no resolution (pass 1) cuts were found	45	30	31	25	21	33	35
Of those nodes for which no resolution cuts were found:							
% for which other cuts were found	33	68	70	78	88	86	62

[a] All pass 3 cuts are unit clauses.

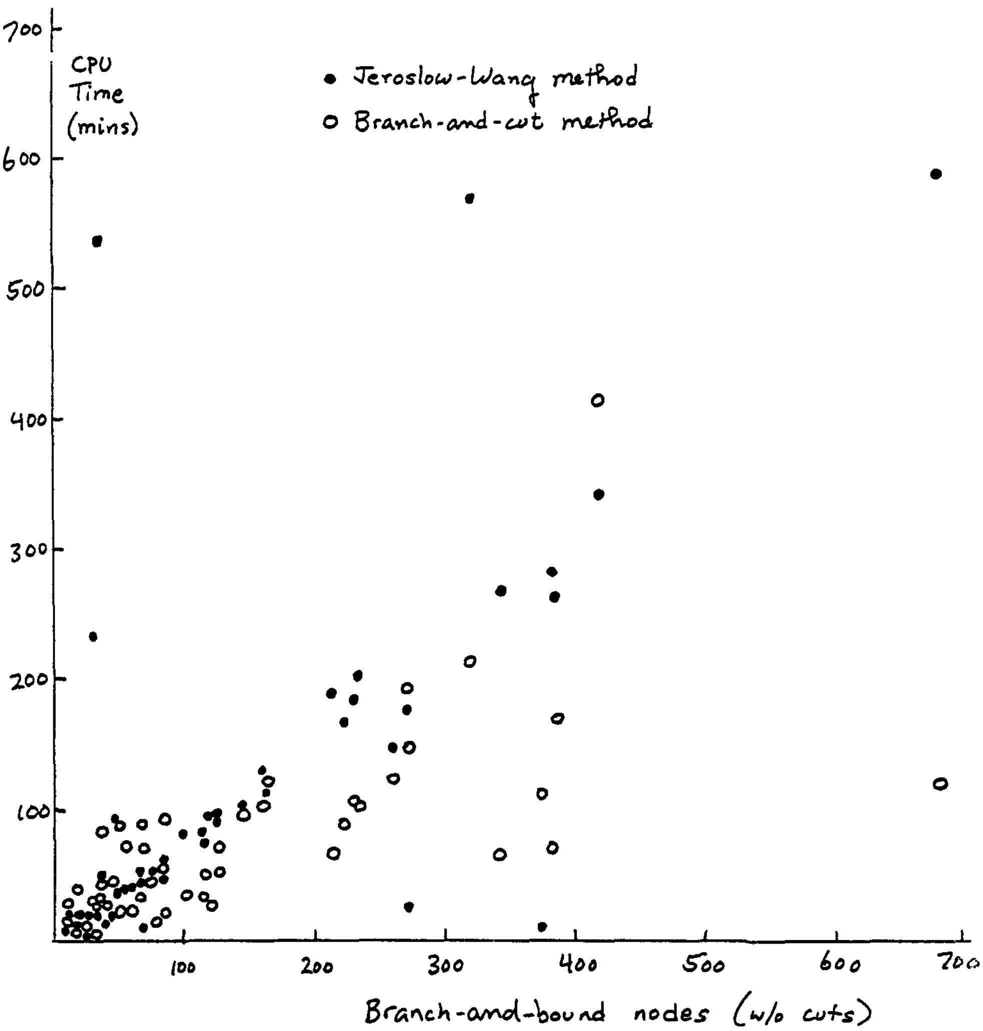

Fig. 1. Comparison of computation times for the Jeroslow–Wang and branch-and-cut methods vs the number of nodes in the branch-and-bound tree.

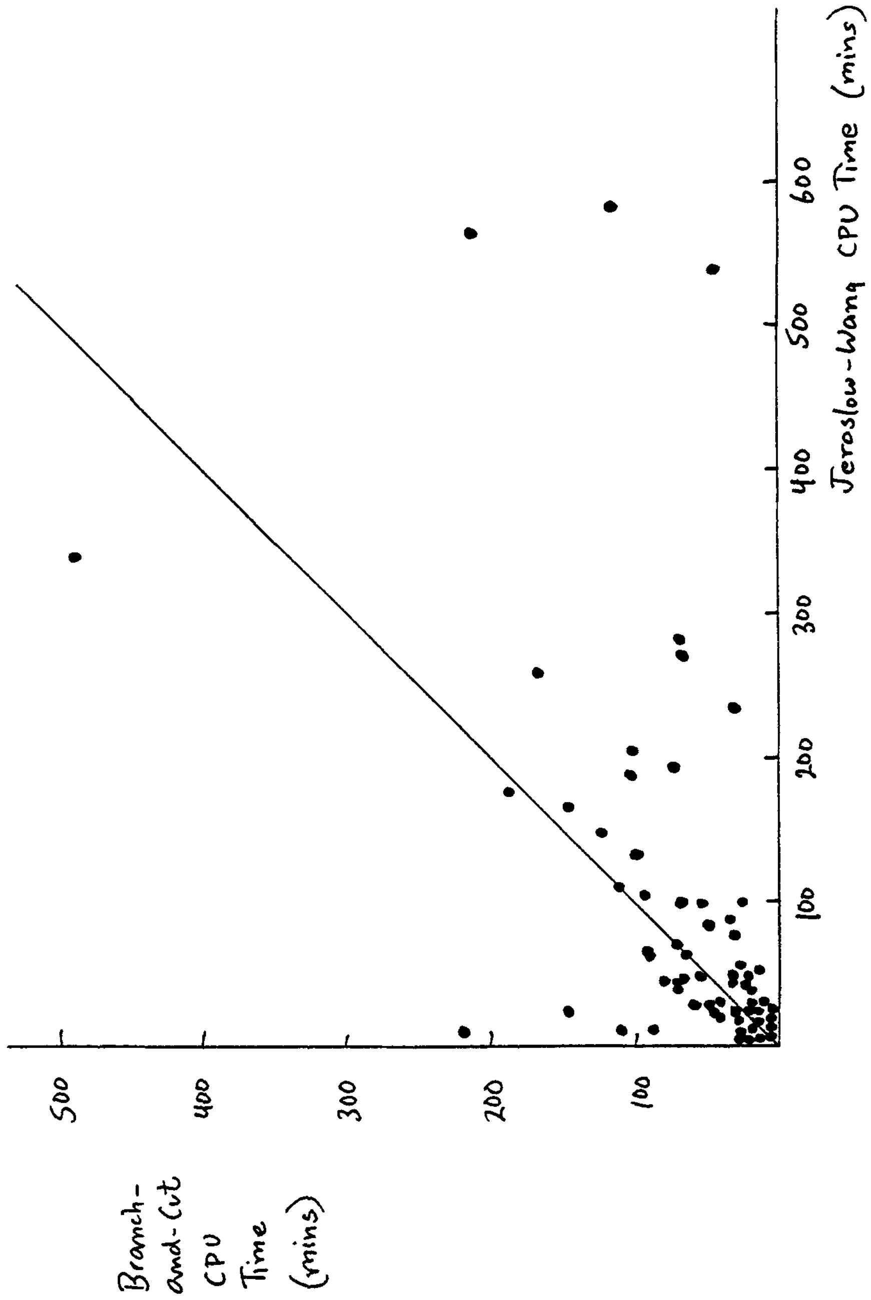

Fig. 2. Computation time for the branch-and-cut method vs that for the Jeroslow–Wang method.

with our FORTRAN implementation of the Jeroslow–Wang method. The variable-fixing and unit resolution subroutines of the latter, which consume nearly all of the computation time, are the same as those used in our algorithm.

As predicted, the difficulty of the problems peaks as they make the transition from satisfiability to unsatisfiability. We also see in the results for 100 variables that the cutting planes are quite effective when the branch-and-bound tree is large. The trees never became large in the problems with 50 variables, and in these problems the cutting planes are not helpful.

Table 2 displays the performance of the separation algorithm on problems with 100 variables. Of all nodes generated in branch-and-cut trees, 67% required cuts (i.e., required that Step 3 of the Branch-and-Cut Algorithm be executed). At the other nodes, the LP relaxation (in Step 2) had either an integer solution or a positive objective function value.

Figure 1 plots computation times for the Jeroslow–Wang and branch-and-cut methods against the number of nodes in the branch-and-bound tree (without cuts). The problems represented are those with 100 variables and 800, 850 or 900 rows. The trend of points shows clearly that the relative advantage of branch-and-cut increases with the size of the tree.

Figure 2 plots the computation times for branch-and-cut against those for Jeroslow–Wang. The problems represented are all those with 100 variables. Branch-and-cut is usually superior when the computation times are large.

The stepwise method was quite effective for solving LP relaxations. For problems with 100 variables and 700 rows, it ran 4.3 times faster than the dual simplex method (applied to the full problem) and 3.9 times faster than the primal simplex method on the 10 initial LP relaxations. For problems with 100 variables and 1000 rows, it ran 6.3 times faster than dual simplex and 4.0 times faster than primal simplex.

9. Conclusions

Since problems that require more branch-and-bound nodes tend to be harder for all three methods tested here, it is meaningful for our purposes to measure the difficulty of a problem by the size of its branch-and-bound tree. Given this, the branch-and-cut method is substantially better for relatively hard inference and satisfiability problems, and the Jeroslow–Wang method substantially better for easier problems. The pure branch-and-bound method is nearly always dominated by one of the other two.

Our extension from resolution cuts to a larger class of cuts is clearly worthwhile for the type of problems solved here, for at least two reasons (table 2). First, it usually permits us to find separating cuts when there are no separating resolution cuts. Second, it leads to the generation of very effective unit-clause cuts in most instances.

The branch-and-cut method acquires a certain amount of direction-finding "intelligence" by solving the LP relaxation and identifying separating cuts, activities that consume substantial overhead. The Jeroslow–Wang method, on the other hand, is directed only by a heuristic that fixes variables in an order more likely to simplify the problem quickly. The branch-and-cut overhead becomes worthwhile when a satisfiability problem is "almost" unsatisfiable (not too many solutions) or "almost" satisfiable (many variables must be fixed before the probability of falsifying a clause is high). In the implementation tested here, branch-and-cut becomes superior when the size of the branch-and-bound tree (without cuts) exceeds 150 nodes or so.

It is likely that this breakeven point can be decreased. Whereas the Jeroslow–Wang method can probably not be much improved algorithmically (aside from the use of parallel algorithms), the branch-and-cut method is bottlenecked by the necessity of solving LP relaxations. The solution of the LP can certainly be accelerated, if only by using one of the faster LP routines now available, in conjunction with our stepwise method. Another alternative would be to replace the stepwise approach with a more sophisticated technique based on a similar idea. One option would be the pivot and probe algorithm of Sethi and Thompson [17], which reduces dual degeneracy by avoiding redundant constraints.

References

[1] C. Blair, R.G. Jeroslow and J.K. Lowe, some results and experiments in programming techniques for propositional logic, Comp. Oper. Res. 13 (1988) 633–645.

[2] S.A. Cook, The complexity of theorem-proving procedures, *Proc. 3rd ACM Symp. on the Theory of Computing* (1971) pp. 151–158.

[3] M. Davis and H. Putnam, A computing procedure for quantification theory, J. ACM 7 (1960) 201–215.

[4] W.F. Dowling and J.H. Gallier, Linear time algorithms for testing the satisfiability of Horn formulas, J. Logic Programming 3 (1984) 267–284.

[5] J. Franco, On the probabilistic performance of algorithms for the satisfiability problem, Information Processing Lett. 23 (1986) 103–106.

[6] J. Franco and M. Paul, Probabilistic analysis of the Davis–Putnam procedure for solving the satisfiability problem, Discrete Appl. Math. 5 (1983) 77–87.

[7] M.R. Genesereth and N.J. Nilsson, *Logical Foundations of Artificial Intelligence* (Morgan Kaufmann, Los Altos, CA, 1987).

[8] J.N. Hooker, Generalized resolution and cutting planes, Ann. Oper. Res. 12 (1988) 217–239.

[9] J.N. Hooker, A quantitative approach to logical inference, Decision Support Systems 4 (1988) 45–69.

[10] J.N. Hooker, Resolution vs cutting plane solution of inference problems: Some computational experience, Oper. Res. Lett. 7 (1988) 1–7.

[11] J.N. Hooker, Input proofs and rank one cutting planes, ORSA J. Computing 1 (1989) 137–145.

[12] R.G. Jeroslow and J. Wang, Solving propositional satisfiability problems, Ann. Math. AI 1 (1990) 167–187.

[13] D.W. Loveland, *Automated Theorem Proving: A Logical Basis* (North-Holland, 1978).

[14] E.R. Marsten, The design of the XMP linear programming library, ACM Trans. Math. Software 7 (1981) 481–497.
[15] M. Minoux, LTUR: A simplified linear-time unit resolution algorithm for Horn formulae and computer implementation, Information Processing Lett. 29 (1988) 1–12.
[16] P.W. Purdom and C.A. Brown, Polynomial-average-time satisfiability problems, Information Sci. 41 (1987) 23–42.
[17] A.P. Sethi and G.L. Thompson, The pivot and probe algorithm for solving a linear program, Math. Programming 29 (1984) 219–233.